Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
08 JAN 2015	**N/A**	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Parallel Software Model Checking	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
Chaki /Sagar	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**3**	

Parallel Software Model Checking

Team Members

Sagar Chaki, Arie Gurfinkel, Derrick Karimi

Project Description (Technical Content)

As the DoD continues to become software reliant, rigorous techniques to assure the correct behavior of programs are in great demand. Software model checking (SMC) is a promising candidate, but its scalability remains unsatisfactory. Recent years have seen the emergence of HPC technologies, e.g., multi-core processors and clusters. Yet, few software model checkers are designed to use this cheap and abundant computing power. A key reason is that model checking is at its core a graph search – where the graph is the state-space of the model – which is difficult to parallelize effectively (i.e., obtain reasonable speedups). The main challenge is to partition the search among the CPUs in a way that limits duplicated effort and communication bottlenecks. A promising approach is to start with a verification algorithm that maintains a "worklist" and to distribute elements of the worklist to different CPUs in a "balanced" manner. New elements are added to the worklist as a result of processing an existing element. For example, this strategy has been used successfully to parallelize the breadth-first-search in the SPIN model checker. This project will explore this strategy to parallelize the generalized PDR algorithm for software model checking. It belongs to TF1 due to its focus on formal verification.

void main() {	$c_1: x = 0 \Rightarrow P(x)$
int x = 0;	$c_2: P(x) \wedge x < 10 \wedge x' = x + 1 \Rightarrow P(x')$
while(x < 10) x++;	$c_3: P(x) \wedge x \geq 10 \wedge x \neq 10 \Rightarrow Error()$
assert (x == 10); }	$Q: Error()$

Figure 1.(Left) a program P; (Right) HSR problem.

Generalized PDR. Generalized Property Driven Rechability (GPDR)[i] is an algorithm for solving HORN-SMT reachability (HSR) problems. A HSR problem consists of a set of HORN-SMT clauses $\boldsymbol{C}$ and a reachability query $\boldsymbol{Q}$. A HORN-SMT clause is a logical implication whose antecedent (a.k.a. body) is a conjunction of terms (some of which are predicates) and whose consequent (a.k.a. head) is a single predicate. The query $\boldsymbol{Q}$ is also a single predicate. The solution to the HSR problem is "UNSAT" if there is an interpretation to all the predicates under which: (i) each HORN-SMT clause in $\boldsymbol{C}$ is valid; and (ii) $\boldsymbol{Q}$ evaluates to false.

We target HSR because a number of software verification projects – that target sequential C code, periodic real-time software, Simulink and Lustre programs etc. – work by reducing their problems to HSR. For example, **Figure 1** shows a program $\boldsymbol{Prog}$ on the left and the HSR problem whose solution is "UNSAT" iff $\boldsymbol{Prog}$ is safe (i.e., does not violate the assertion, which is the case) on the right. Note that there are three clauses -- $\boldsymbol{c_1, c_2, c_3}$ – and the predicate $\boldsymbol{P(x)}$ represents the loop invariant. Thus, an effective parallel solution to HSR will be of immediate benefit to these projects. It will also establish the SEI as an important player in the budding HSR solving community.

Task 1. Develop parallel GPDR algorithm. GPDR is naturally worklist based. Each element of the worklist is a triple (P, σ, n) where P is a predicate, σ is a context (consisting of logical formulas representing known facts or lemmas) and n is a search depth. Informally, the item represents a question of the form "is the satisfiability of P derivable in n steps under the assumption σ"? Processing (P, σ, n) involves the following steps for each clause c whose head is P: (Step1) solve an SMT query q constructed from the body of c and σ;

(Step2) if q is UNSAT, update σ, and return result UNSAT; (Step3) if q is SAT generate new worklist sub-items $(P_i, \sigma, n-1)$ for each predicate P_i appearing in the body of c. If the result of processing each of these sub-items is SAT, return result SAT. If the result of at least one sub-item is UNSAT, either generate new sub-items or return UNSAT if no further sub-items can be generated. We will develop parallel GPDR in 3 stages:

1. Stage 1. Managing dependencies between items in a provably correct way. As seen above, the result for an item depends on those of sub-items. The dependency forms a directed acyclic graph (DAG) since one item can be a sub-item of two other items. Version 1 of parallel GPDR (PGPDR) will overlay this dependency management on top of distributing the items to CPUs. We will prove correctness of our algorithm.

2. Stage 2. Terminating "junk" queries. Consider the sub-items $\{(P_i, \sigma, n-1)\}$ generated in Step3 above. As mentioned above, if the result of even one of these sub-items is UNSAT, then the results of the other sub-items become irrelevant. PGPDR version 2 will detect and terminate such obsolete queries. Since item dependencies form DAGs, this will require reference counting to avoid premature "garbage collection".

3. Stage 3. Minimizing results. In the final stage, we will develop algorithms to minimize the learned lemmas. Since the logics we will operate over (linear real arithmetic, bit-vectors etc.) do not have canonical forms, we expect that over time our lemmas will become syntactically redundant and also irrelevant. Minimization will reduce this redundancy and eliminate useless lemmas periodically.

Task 2. Design scalable architecture for PGPDR. We will scale PGPDR from multicore CPUs to clusters. Late binding and a layered architecture will allow variation in runtime framework and data storage. We will develop a task control and message-passing API, and use them to build and deploy PGPDR. A data abstraction layer will also be included. It will allow the choice of data store to evolve based on observed data access requirements of the algorithm. We will target two deployment modes: single node and clustered. As the implementation of the algorithm matures, our architecture will handle increasingly large problems. Initial analysis looked at candidate openly available software components on which to build PGPDR in both deployment modes.

1. Single Node. Initially, PGPDR will be deployed on a single multicore machine. The task control will be implemented with a thread pool, messaging and caching. The item-dependency DAG will be stored using a graph database.

2. Clustered. Subsequently, PGPDR will be deployed to a cluster. We believe that the algorithm's layered architecture will allow it to be easily ported to common cluster-scale runtime frameworks, which supports the task and messaging architecture in our PGPDR design) and distributed databases.

Evaluation. We will evaluate the parallel GPDR by comparing it to sequential GPDR and measuring speedup as a function of the number of cores.

Related Work. Both the LTSmin[ii] and SPIN[iii] projects have developed algorithms for multi-core LTL model checking. These algorithms are explicit-state and target modeling languages such as PROMELA and DVE. In contrast, our algorithm is symbolic and targets HORN-SMT reachability. Ditter et al.[iv] have developed GPGPU algorithms for explicit-state model checking. Our project will target multi-core CPUs and compute clusters where the programming model is different (e.g., task-oriented and not restricted to SIMD).

Team. Sagar Chaki and Arie Gurfinkel are experts in software model checking and the GPDR algorithm. Derrick Karimi is experienced in high-performance and distributed computing, and a lead developer of ETC[v].

[i] Krystof Hoder, Nikolaj Bjørner: Generalized Property Directed Reachability. SAT 2012: 157-171
[ii] Alfons Laarman. *Scalable Multi-Core Model Checking*. 2014. PhD Thesis
[iii] Gerard J. Holzmann: Parallelizing the Spin Model Checker. SPIN 2012: 155-171
[iv] Alexander Ditter, Milan Ceska, Gerald Lüttgen: On Parallel Software Verification Using Boolean Equation Systems. SPIN 2012: 80-97

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0002056